Software Project: *Text Adventure – Computerspiel*
Adviser: Alexander Koller
Summer Semester 2001

# Documentation for the Actions Module

# Part I:

# Interaction between Main Functor and Database

Student: Garance Paris
30th September 2001

Computational Linguistics
Department 4.7 General Linguistics
University of the Saarland

*This document describes how the actions' main functor (*`Execute.ozf`*) uses the action database in the text adventure game which was implemented during summer semester 2001 as a student software project at the Computational Linguistics Department of the University of the Saarland. The functors and files described are the* `ActionDatabase.oz` *in itself and the functors which make use of it: syntax of the STRIPS-like rules found in the database, loading of the rules at the start of the game (functor* `InitActions.ozf`*), preprocessing of the input to the actions module (functor* `Preprocessing.ozf`*), and beginning of* `Executable`*, the function which tests an action's precondition and possibly executes the action, up to the lookup of the actions in the database during the game and evaluation of those terms in the rules whose value is world state-dependent (both in functor* `LookupAction.ozf`*).*

## Contents

# 1. Introduction

This paper is the first part of the documentation of the actions module in the text adventure game to whose implementation I contributed during summer semester 2001 at the Computational Linguistics Department of the University of the Saarland.

The functors and files described are those which the actions module's main functor, `Execute.ozf`, calls before it can start the process of deciding whether an action can be executed in the present state of the world. As it happens, these functors and files are also the ones which involve an interaction between the main functor and the `ActionDatabase.oz`.[1] The main functor itself, however, will not be the object of this paper; instead we refer the reader to the second part of this documentation, written by Ruth Kusterer.

Overall, the actions module is responsible for executing whatever actions the player typed in – or at least, whatever the preceding linguistic modules understood the player meant. In a virtual world such as ours, this comes down to turning a before-action world state description into an after-action world state description.

Our module takes its input from the resolution module, and in the end, its output is turned over to the content determination module.[2] Inherently, the input is a semantic description of what the player wants to do and what individuals in the world this should involve. Thus for example, if the player typed in *"Close the door"*, the input for the actions module would be something like `close(patient:door1)`.

However, this input alone does not carry enough information to allow the execution of the action: we also need a description of how the before-action state of the world is to be changed by the given action into an after-action state, and under what conditions this action should be allowed to take place. In order to represent this additional information for each verb necessary in our text adventure game, we defined rules for the actions in the `ActionDatabase.oz`, using a variant of what is commonly called the STRIPS language.

**Plan of the Paper** Section 2 thus starts out by briefly reviewing how actions were represented in the traditional STRIPS language (2.1.1), before we then go on to present how we adapted it to our own specific needs (2.1.2). Following this, section 2.2 shows how we coded these action descriptions in Oz (file `ActionDatabase.oz`), and section 2.3 introduces two additions we made to the STRIPS language in order to increase the rules' expressive power: an alternative syntax for the terms of the rules, and functional terms whose value is world state-dependent.

The remaining sections of the paper are then structured according to the time at which the various functions of the functors are called during the game. Section 3 is concerned with the initialisation of the database at the beginning of the game (functor `InitActions.ozf`), and the object of section 4 is the manner in which actions are looked up in the database throughout the game (functor `LookupAction.ozf`), with special attention being given to the implementation of the `LookupInDB` function.

The next two sections (4.3 and 5) then show how the more expressive terms mentioned above are dealt with, some of them just after lookup time and some of them a bit later at execution time, and finally, as an add-on, the appendix shows how to increase the flexibility of the way in which the input is matched with the database entries, by solving a problem with open records we had with the most recent version of module.

[1] For information concerning the action database, the reader should also consult the more specific documentation by Mario Cattaneo.

[2] See the general architecture of the system at `http://www.coli.uni-sb.de/cl/courses/spiel-01/handout9-4.ps`.

# 2. Representations for Actions

## 2.1. Using the STRIPS Language

### 2.1.1. The Original STRIPS rules

Our action database was written in a variant of what is referred to as the **STRIPS language**, named after the Stanford Research Institute Problem Solver, a planner developed in 1971 by the organisation that is now called SRI International. It was for STRIPS that Fikes and Nilsson (1971) first devised a formal way of representing actions, and although the STRIPS planner is outdated today, the syntax that was originally used to define its rules[3] continues to be used to describe actions, whether in the realm of planning or in other applications such as in our text adventure game (Russell and Norvig, 1995).

In STRIPS, the rules representing actions consisted of the following elements:

- an action descriptor: a verb used as a "name" for the action
- arguments: (an ordered list of) variables standing for individuals in the world model
- a precondition: a conjunction of literals that must be true before the action can be executed
- an effects list, often divided into a delete list and an add list: conjunctions of literals that describe how the situation changes when the action is executed

For example, in a simple world made of blocks which can be stacked onto each other, if we wanted a robot to pick up a cube, we might write the following rule:

```
pickup(x)
Precondition & Delete List:  ONTABLE(x), CLEAR(x), HANDEMPTY(h)
Add List:  HOLDING(x)
```

In order to execute an action, the arguments of the corresponding rule must be instantiated, and the result is what we call an **operator**. Any remaining variables in the precondition or effects list of such an operator are assumed to be existentially quantified (such as the $h$ in the above example: at least one of the robot's hands must be empty), and there is an additional well-formedness condition stipulating that any free variables occurring in the add or delete lists must also occur in the precondition (Wahlster, 2001).

Given the before-action world state description, we can use such an operator to produce an after-action world state description by first deleting all the literals in the delete list from the current description of the world, and then adding all the literals in the add list. All literals not mentioned in the delete list carry over to the after-action world state description: this is called the **STRIPS assumption** and is one way of dealing with the frame problem[4] (Nilsson, 1998).

[3]Nilsson (1998) calls uninstantiated STRIPS operators **rules**, whereas Russell and Norvig (1995) refer to them as **operator schemata**. Wahlster (2001) does not use different terms for instantiated and uninstantiated operators. I have opted for the terminology as it is used by Nilsson (1998) for the sake of clarity.

[4]The frame problem is concerned with the question of what should happen to those statements describing the world (in our text adventure game, the assertions in the ABox) that are left untouched when we execute the action at hand. In other words, the problem is to determine how the context of a given change should be treated in a dynamically changing environment (Ingargiola, 1995). A good summary of possible approaches to the frame problem is given by Lormand (1999).

| close(patient: x) | | |
|---|---|---|
| Precondition: open(x),accessible(x) | | |
| Effects: | Add List: closed(x) | |
| | Delete List: open(x) | |
| User Knowledge: | | Add List: closed(x) |
| | | Delete List: open(x) |

Figure 1: The Rule for `close`

### 2.1.2. Our STRIPS-like Rules for Actions

In order to define the precondition and the effects lists for each of the verbs necessary in our text adventure game, we adapted the traditional STRIPS rules in the following way (see example in figure 1):

- Instead of an ordered list of argument variables, the variables are assigned to a specific thematic role such as patient, target, etc., thus using the same syntax as that of the input coming from the resolution module (see introduction). The agent of the actions being always the player himself, the agent role is not mentioned.

- But the main difference is our use of two different kinds of effects lists, the first one for changes affecting the world state description and the second for those affecting the player's world knowledge. The slot for the first kind of changes, i. e. the effects proper, is plainly termed **effects** (as in STRIPS), whereas the slot containing effects on the player's representation of the world is termed **user knowledge**.[5]

The reason we needed two different kinds of effects is that the language processing modules do not make use of the entire world representation, but instead only of those elements which the player knows about: in the resolution module, for one, referring expressions can only refer to individuals which the player knows to exist (Gabsdil et al., 2001).

For this reason there are two ABoxes in the game, a world ABox and a user knowledge ABox, the second one being usually a subset of the first, both of which must be updated when an action is executed. The actions module itself, however, concerns itself only with the update of the world ABox, the update of the user knowledge being deferred to the content determination module (see respective documentation).

Although in the `close` rule given above, as in most rules, both of the ABoxes are updated in the same way, this is not always the case. Indeed, the separate processing of changes affecting the world and changes affecting the user knowledge is useful in two ways. First, in the context of an adventure game, it may sometimes be desirable for an action to have effects the user may not be aware of (e. g. pressing a button may cause something to happen in some faraway room). But more importantly, some actions, such as a `look` action, must trigger a description to the player of his immediate surroundings and an extension of the player's knowledge about the world, something which is achieved by the introduction of a special term, `describe(x)`, into the user knowledge effects list, which is left untouched by `Execute.ozf` and passed on to the content determination module.

[5] In this paper, I use *effects lists* to refer to either the effects lists proper or to both kinds of effects; it should be clear from the context whether at any point one or both lists are referred to.

```
entry(action: close(patient:1)
      pre:[open(1)
           accessible(1)]
      effects: effects(add:[closed(1)]
                       delete:[open(1)])
      uk: uk(add:[closed(1)]
             delete:[open(1)]))
```

Figure 2: The Database Entry for `close`

## 2.2. The Syntax in `ActionDatabase.oz`

### 2.2.1. Representing the Rules as Oz Records

The STRIPS-like rules needed for the game are defined in the file `ActionDatabase.oz` (see also documentation by Mario Cattaneo), using Oz records similar to the one shown in figure 2. Each of these has `entry` as its label and carries four features `action`, `pre`, `effects` and `uk`, whose respective values are described below.

`action` Value: An embedded Oz record, with the action descriptor for label and thematic roles for features. In section 2.1.2, we saw that the value of these thematic roles corresponds to the argument variables in the original STRIPS rules, which stand for individuals in the world model.

`pre` (precondition) Value: A list representing the conjunction of terms which must be true for the action to be executable, as explained in section 2.1.1. We will come back to the syntax of terms in section 2.3.1.

`effects` and `uk` (user knowledge) Value: Both Oz records with the respective labels, carrying two features `add` and `delete` followed by a conjunction list of terms, which represent the elements to be added to the respective ABox and the elements to be deleted.

### 2.2.2. Marking Co-Reference with Integers

Although the value of the thematic roles in the `action` record will later need to be unified with individuals of the world model when the rules are instantiated, at rule-writing stage co-reference is indicated by using integers instead of Oz variables, as suggested by Duchier et al. (2001a), thus avoiding the problem of having to declare Oz variables locally within the rule. These integers will later be replaced by Oz variables, as we will see in sections 3 and 4.2.

This implementation has the advantage that the rules are less cumbersome then they were in earlier versions of the `ActionDatabase.oz`. Indeed, previously, we attempted to use Oz variables instead of integers, and declared the variables implicitly by introducing them in the pattern of a case statement, as shown in figure 3.[6]

However, this meant that we had to surround each and every rule with a case statement. Moreover, in order to declare the `Input` variable which heads the case statement, we had to introduce it as the argument of a function and write this, too, into every rule, instead of

[6] Note that the early format of the rules shown in figure 3 relied on an obsolete version of the `LookupInDB` function, not on the one presented in section 4.

```
fun{$ Input}
   case Input of open(patient:X)
   then entry(action: close(patient:X)
              pre: [open(X)
                    accessible(X)]
              effects: ...
              uk: ...)
   else nil end
end
```

Figure 3: Early Format of Database Entries

declaring once and for all a function `Encapsulate` which could be applied onto each rule, as we did in later versions of the module (see section 3).

What's more, in order to avoid raising a runtime error ("`Missing else clause`") at the evaluation of the rules if the `Input` did not unify with the only pattern of the case statement, the encapsulating function also needed to have some return value in this case (here `nil`). It is the use of integers in later versions of the database which allowed us to simplify in various places the format of the rules, thus making the writing of the rules much simpler than in the early implementations.[7]

## 2.3. More Powerful Terms for the Precondition and Effects

### 2.3.1. Embedding Description Logic Syntax in the STRIPS Language

In the example in section 2.1.1, we gave an illustration of what the terms of the precondition and effects lists looked like in the original STRIPS language. These terms could represent either properties of world individuals or relations existing between these individuals. Thus in the very simple block world mentioned in section 2.1.1, `ONTABLE(x)` might have meant that $x$ is on the table, and `ON(x,y)` that cube $x$ is stacked on top of cube $y$.

However, in our text adventure game, the virtual world surrounding the player is represented by using the RACER description logic system (Haarslev and Möller, 2001), with which we communicate through a socket, `racerlink.ozf` (written by Alexander Weber), that formulates properties and relations in the world in a manner different from that of the STRIPS language: to say that $x$ is on the table, `racerlink.ozf` writes `instance(x ontable)`, whereas the relation mentioned above would be expressed as `related(x y on)`, in which $x$ and $y$ are individuals of

[7] Moreover, although we do not make use of this in the present version of the database, the new implementation would also allow in the precondition of the rule implicitly existentially quantified variables which do not appear as arguments of the verb.
This could actually be useful to deal with some kinds of verbs, such as *"Stand up"*, in which it seems fairly unnatural to obligatorily require the player to add "from where" (*"Stand up from the chair"*), since we would assume the player to correctly remember where he is sitting. Indeed, in order to accept such input, we would need a rule requiring the current location of the player to be some kind of seating which remains unspecified in the input, thus resulting in the an illegal entry if we used Oz variables, because the variable $X$ is not declared anywhere:

```
entry(action: 'stand-up'
      pre:['has-location'(myself X)
           seating(X)]
      effects: ...
      uk: ...)
```

the world, `ontable` is what we call a **concept**, and the relation `on` is termed a **role** (Wahlster, 2001).

Moreover, in addition to expressing relation and properties differently, description logics such as RACER are also more powerful than the STRIPS language: they allow to state facts more complex than those which the STRIPS language can describe. For example, it is possible to negate concepts, as in `not(inventory-object)` or to express equality between individuals, by writing `equal(x y)`.

In order to allow maximum flexibility in writing the rules in `ActionDatabase.oz`, we wanted to allow both the STRIPS and the description logic syntax. Internally however, the functions in `Execute.ozf` use the more complex description logic variant, and conversion of any literals from the STRIPS language to the description logic syntax is done by a `Convert` function, which is applied onto the output of the `LookupInDB` function, both of which will be described in section 4.

### 2.3.2. The Need for Functional Terms

We also saw in section 2.1.1 that in the original STRIPS language, the elements of a precondition or effects list were literals, i. e. function free terms. However, this is not the case in our database entries, for behind some special terms, we hid function calls which send off queries to RACER (see also the documentation of the `ActionDatabase.oz`, by Mario Cattaneo).

The reason we introduced these functional terms is that they enable us to have some of the elements in the precondition and effects lists depend on the state of the world at the time the action is executed. For example, in the case of the rule for `drop`, we want to express the idea that if the player drops, say, a key, the key will fall *wherever the player is* when the player drops it, and stay located there until something else happens to it.

The two functional terms we introduced are actually calls to two corresponding RACER functions (Haarslev and Möller, 2001, pp. 57–58). They are:

- `concept-instance(C)` which returns the individual that is an instance of C, and
- `individual-filler(I R)` which returns the individual that stands in a relation R with the individual I

In the `drop` example above, we can thus now use the first of these to write `concept-instance(here)` to refer to the present location of the player. An example for the use of `individual-filler` would be in the rule for `take`[8], in which the patient is deleted from its previous location and instead relocated "on" the player, as part of his equipment:

```
entry(action: take(patient:1)
      pre:[not(inventory-object(1)) ...]
      effects: effects(add:[related(1 myself has-location)]
                       delete:[related(1
                                       individual-filler(1 has-location)
                                       %% previous location of the toolbox
                                       has-location)]))
```

Of course, by the time the action is actually executed, these functional terms must have been evaluated and replaced by their value. This condition comes in addition to the one which stipulates that the arguments of the rule must have been instantiated, which we mentioned in section 2.1.1. This evaluation is the job of the function `Evaluate`, which we will present in section 5.

[8] The content of the user-knowledge lists in this rule being the same as the content of the effects, it has been omitted here.

```
for Entry in Entries % from ActionDatabase.oz
do
   fun{Encapsulate Entry}
      fun{$} {SubstituteInts Entry} end
   end
   EncEntry={Encapsulate Entry}
   Key={Record.label Entry.action}
in
   ActionDB.Key := EncEntry|{Dictionary.condGet ActionDB Key nil}
end
```

Figure 4: Excerpt from `InitActions.oz`

# 3. Initialising the Database

At the start of the game, the rules defined in `ActionDatabase.oz` are loaded into an Oz dictionary by the functor `InitActions.ozf` (see figure 4). However, before being added to the dictionary, each rule is first **encapsulated**, meaning that it is made to be the return value of a function, so that in the future, each time we want to apply a rule, we can, by calling the corresponding function, get a new “copy” of the rule, containing fresh, uninstantiated variables, with the encapsulated entry remaining intact for future uses of the same verb.

In addition to encapsulating it, we also surround each rule by a call to `SubstituteInts` (see figure 5), a function which will later, at each evaluation of an encapsulated rule, create a brand new variable assignment (i. e. an Oz dictionary to keep track of the mapping between integers and variables) and then substitute a variable for each integer, declaring as it goes any newly introduced Oz variables.

The entries in the Oz dictionary `ActionDB` are thus anonymous functions which, when later evaluated, will turn a rule with integers into a rule with Oz variables. These entries are loaded into the dictionary, using as keys the rules' action descriptors (i. e. the labels of the Oz record in the `action` field).

However, since for some verbs, there may be several entries differing in either the thematic roles required by the verbal relation (in the case of a verb allowing optional arguments) or in the content of their precondition and effects lists (e. g. if the player kisses something ugly, he will be disgusted, but not if it is nice-looking), each key in the dictionary actually corresponds to a list of rules.

Finally, after the entries are loaded, the Oz dictionary containing the encapsulated rules is exported by the functor, ready to be imported by the functor `LookupAction.ozf`, which will be described in the following section.

# 4. Looking up Actions During the Game

## 4.1. The Call on `LookupInDB` by `Execute.ozf`

During the game, when the player tries to execute an action, the input we receive from the preceding modules is a list of lists of Oz records. The outer list represents a disjunction of possible parses of the player input (in the case of a syntactic ambiguity) and the inner list a conjunction or succession of actions within one parse. In the Oz records, all referential entities have previously been resolved to individuals of the world model by the resolution module. For

```
fun{SubstituteInts Entry}
   VarAss={Dictionary.new}
   fun{ToVar Int}
      if {Not {HasFeature VarAss Int}}
      then X in
         VarAss.Int := X
         X
      else VarAss.Int
      end
   end
   fun{Substitute Term}
      if {IsInt Term} then {ToVar Term}
      elseif {Atom.is Term} then Term
      elseif {Record.is Term} then {Record.map Term Substitute}
      end
   end
in
   {Substitute Entry}
end
```

Figure 5: The function `SubstituteInts`

example, if the player typed in *"Unlock the toolbox with the key and open it"*, we might get the following input (in which `toolbox2` would be a "toolbox-with-key"):

```
[
 [unlock(patient:[toolbox1] instrument:[key1]) open(patient:[toolbox1])]
 [unlock(patient:[toolbox2]                  ) open(patient:[toolbox2])]
 [unlock(patient:[toolbox1] instrument:[key1]) open(patient:[key1])]
]
```

The reason the value of each thematic role is a list is that in the case of a plural noun phrase or of a conjunction in the input, such as in *"Open the toolboxes"* or *"Open the toolbox and the chest"*, we would respectively get

```
    [[open(patient:[toolbox1 toolbox2])]]
and [[open(patient:[toolbox1 chest1])]]
```

But since the `action` records in the database are instead of the form *VerbalRelation(ThematicRole:ArgumentVariable)* (see section 2.2.1), before `Execute.ozf` can call the `LookupInDB` function for the actions, we first have to apply a preprocessing step, in which noun phrase coordinations are replaced with sentence coordinations by "flattening"[9] out the NPs. For the first example above, we would thus obtain:

```
[[open(patient:toolbox1) open(patient:toolbox2)]]
```

After this, all elementary actions are looked up in the database by mapping of the `LookupInDB` function onto the list of lists of records, and they are converted into the more powerful syntax mentioned in section 2.3.1 by the `Convert` function, which will be described in section 4.3:

[9] In the case of more complex input, the flattening out involves a process of distribution, so that `[[ a(p:[w x] i:[y z]) ]]` becomes `[[ a(p:w i:y) a(p:w i:z) a(p:x i:y) a(p:x i:z) ]]`. It seems however that in some cases this could pose a problem: if the input was *"Unlock the toolboxes with the keys"*, since it is probable that one key fits each toolbox, we would not want to obtain a conjunction of four actions,

```
Entries={Map FlatInput
         fun{$ Disjunct}
            {Map Disjunct
             fun{$ Conjunct}
                {Convert {LookupInDB Conjunct}}
             end}
         end}
```

However, as we mentioned in section 3, each key in `ActionDB` actually corresponds to a list of verbs. For this reason, the output of mapping the `LookupInDB` function onto the flattened input returns a list of lists of lists of entries, in which the most embedded list represents the disjunction of possible entries in the action database.

Consequently, before going on to evaluate whether the input of the player is executable (which is the job of the module's main functor, `Execute.ozf`, and is the object of the second part of this documentation, authored by Ruth Kusterer), we bring this back to a disjunction of conjuncts (a list of lists of records), by distributing and associating according to common logic, making use of the equivalence between the following two lines:

$$(((A \vee B \vee \ldots) \wedge (C \vee D \vee \ldots) \wedge \ldots) \vee Disjunct2 \vee \ldots) \Leftrightarrow$$

$$((A \wedge C) \vee (A \wedge D) \vee (B \wedge C) \vee (B \wedge D) \ldots \vee Disjunct2 \vee \ldots)$$

## 4.2. Looking up Elementary Actions

When the `LookupInDB` function is called by `Execute.ozf` on each conjunct of the input, the first step is to retrieve from the database the list of encapsulated rules which correspond to the label of the Oz record (see figure 6). Then, once we have the list, we need to select from it those rules which actually unify with the input record.

The problem here is that we cannot just proceed by trial and error, unifying each rule in turn with the input and collecting those which do in a list, because failed unifications in Oz result in errors, be they static analysis errors, e. g. when the system notices that the two records have differing arities, or runtime errors, when it is the telling to the constraint store which fails (Duchier et al., 2001a, chapter 11).

Had the errors always occurred at runtime, we could have caught the exceptions with `try...catch...end`, but how were we to deal with both kinds of errors? Fortunately, Oz provides a mechanism to deal with such cases, in which failed unifications should not be considered as a programming error, but as logical information meaning that the equation is unsolvable: Duchier et al. (2001a) suggest to use Oz's encapsulated search.

All search engines in Oz have this in common that they take a unary predicate as their input and return a list of one or more solutions to this predicate. If no solutions are found, `nil` is returned (Duchier et al., 2001b). In particular, `Search.base.one` returns a singleton list containing the first solution found, which in our case is also the only solution. Thus it is this search engine which we use in order to return, for each encapsulated rule in the list we retrieved, the result of its unification with the input record, or `nil` in the case that they do not unify.

```
[[ unlock(patient:toolbox1 instrument:key1)
   unlock(patient:toolbox2 instrument:key2)
   unlock(patient:toolbox1 instrument:key2)
   unlock(patient:toolbox2 instrument:key1) ]]
```

but probably instead two disjuncts, as in:

```
[[ unlock(patient:toolbox1 instrument:key1) unlock(patient:toolbox2 instrument:key2) ]
 [ unlock(patient:toolbox1 instrument:key2) unlock(patient:toolbox2 instrument:key1) ]]
```

```
fun{LookupInDB Input}
   Key={Record.label Input}
   EncapsulatedRules=ActionDB.Key
in
   {Map EncapsulatedRules
    fun{$ DBEntry}
       {Search.base.one
        fun{$}
           Entry = {DBEntry}
           Entry.action = Input %Unification
        in
           Entry
        end}
    end}
end
```

Figure 6: `LookupInDB` function using a Search Predicate

Within the search predicate, we first evaluate the encapsulated rule, `DBEntry`, thereby triggering a call to the function `SubstituteInts`, which we wrote into the body of the encapsulation function when we loaded the rules into the database (see section 3). The result, which is bound to the local variable `Entry`, is thus a local "copy" of the encapsulated rule, but contains Oz variables instead of integers.

Following this, we unify[10] the input record with the content of the `action` field in the rule, thus resulting in the instantiation of the variables in the rule, and give back the modified `Entry` (which, being instantiated, has now become what we called an operator in section 2.1.1) as a solution to the search predicate.

Let us now take a closer look at the instantiation process, by illustrating it with an example. Take the entry given below:

```
entry(action: close(patient:X)
      pre:[open(X)
           accessible(X)]
      effects: effects(add:[closed(X)]
                       delete:[open(X)])
      uk: uk(add:[closed(X)]
             delete:[open(X)]))
```

Imagine we then did the following unification: `close(patient:X)=close(patient:door1)`. Since the $X$ in the `action` record also comes up in the precondition and effects lists, the binding

[10] Note that in the version of the module described here, it is necessary that the input record match exactly the `action` record of the database entry. At the conceptual stage, however, our intuition had been that it should be possible to "ignore" superfluous thematic roles in the input: `kill(patient:worm1 instrument:sword1)` should unify also with `kill(patient:X)`, because although you do not necessarily need a sword to kill such small animal as a worm, it would not be wrong to use one.
In order to achieve this, we tried using open records in the database entries, i. e. records which do not have a fixed arity, and which are denoted in Oz by "..." in the record, such as in `kill(patient:X ...)`. Unfortunately, this lead to difficulties at lookup time which we could not pin down. In November 2001, Dr. Denys Duchier kindly explained to me where the trouble came from, and a suggestion is made in the appendix on how the problematic procedures could be reimplemented to allow the relaxation of the unification constraint.

of the variable $X$ to `door1` would automatically extend to the whole entry, and thus `Entry` would then become:[11]

```
entry(action: close(patient:door1)
      pre:[open(door1)
           accessible(door1)]
      effects: effects(add:[closed(door1)]
                       delete:[open(door1)])
      uk: uk(add:[closed(door1)]
             delete:[open(door1)]))
```

One more thing is important to note when considering the `LookupInDB` function, namely that the evaluation of the encapsulated rule and the binding of the result to a fresh variable takes place *within* the search predicate, because search predicates in Oz cannot refer to free global variables (Duchier et al., 2001a, section 12.5).

In the present case, both `DBEntry` and `Input` are global variables, but both of them are bound before the search: `DBEntry` is a unary predicate and `Input` is a record which does not contain any free variables. Thus we may refer to them inside the search predicate. On the contrary, at the time it is introduced, `Entry` contains free variables which will then become bound on the next line, and consequently it must be declared as a local variable within the search predicate.

### 4.3. Conversion from the STRIPS Language to Description Logic Syntax

As we saw in section 4.1, after each elementary action is looked up in the database, we then convert those terms in its precondition and effects lists which are in STRIPS syntax into the description logic syntax used by `Execute.ozf`.

This is the job of the `Convert` function, which is also part of the `LookupAction` functor. This function deals in turn with each element of the five conjunction lists in the rule (precondition plus add and delete lists of both effects and user knowledge), doing the following:

- the functional terms mentioned in section 2.3.2, as well as any of the content determination terms with the `describe`-label (see section 2.1.2) are passed on untouched;
- records already in description logic syntax, namely those with the labels `instance`, `related`, `equal` and `not`, are left untouched, and in the case of embedded terms, recursion is applied;
- records with a single feature, such as `open(x)`, are converted to `instance(x open)`, and recursion is applied in case of embedding;
- records with two features, such as the `on(x,y)` relation we mentioned earlier, are converted to `related(x y on)`, with recursion also being applied if necessary;
- anything else (constants, other description logic formulae, such as `all(...)` or `some(...)`, etc.) is left untouched.

## 5. Evaluating the Functional Terms

Finally, after having looked up each action in the database and having converted it to the description logic syntax used internally by `Execute.ozf`, there remains one thing to be

[11] Actually, only those variables are instantiated which are arguments of the verb. If there were any existentially quantified variables in the precondition of the rule (see footnote in section 2.2.2), they would not be instantiated at this stage.

done before the checking of the action's precondition prior to its execution can take place: the functional terms contained in the action's precondition and effects (see section 2.3.2) must be replaced by their value.

This is done by the function `Evaluate`, which, for each element of the conjunction list (precondition or effects list) it is receives as argument, recursively looks at the label of the record to determine whether it is a functional term, and if necessary calls up the corresponding RACER function, through the socket `racerlink.ozf` which was implemented by Alexander Weber.

In the latest version of the module, the function `Evaluate` is part of the `LookupAction.ozf` functor, because the evaluation of the functional terms was originally done by the `Convert` function right after the looking up of actions in the database. However, this function is now applied later: within `Execute.ozf`, it is called on the terms of the precondition by `LookupLeftToRight`, which checks whether each element of the precondition is satisfied, and on the terms of the effects lists by `CheckExecuteConjunct`, which attempts to execute each conjunct in a conjunction list (see section 4.1), in the case that the action's precondition is satisfied (both of these functions are described in part two of this documentation, written by Ruth Kusterer).[12]

The reason why we had to postpone the call on `Evaluate` is best illustrated by an example. Imagine the player typed in *"Sit down on the couch and sit down on it"*. Intuitively, in this sequence, the second action should fail, since you can't sit down on something you are already seated on. This is expressed by the precondition of the rule for `sit-down`, which, after instantiation, would look as follows:[13]

```
entry(action: sit-down(target:couch1)
      pre:[seating(couch1)
                   accessible(couch1)
                   not(equal(individual-filler(myself has-location) couch1))]
      effects: effects(add:[has-location(myself couch1)]
                       delete:[has-location(myself concept-instance(here))]))
```

Consequently, the functional term in the precondition of the second `sit-down` action, `individual-filler(myself has-location)`, should only be evaluated *after* the execution of the first action, after which the player is located on the couch. Thus we see that in sequences in which the second action either builds on the effects of the first, or is ruled out by these effects, it is important that the functional terms be evaluated for one action at a time.[14]

[12] In reality, it would probably make more sense to group into one functor both `Evaluate` and `LookupInWorld`, since these are the two functions of our module which interact with RACER.

[13] Once more, the uk slot has been omitted here, because it is exactly similar to the effects.

[14] In addition, remember how we mentioned in the footnote on page 4 that in later versions of the `ActionDatabase.oz`, the use of integers in the rules allows variables not forthcoming in the arguments to be present in the precondition, as it was in STRIPS (see section 2.1.1). Although we do not presently make use of this possibility, should it be exploited in future versions, the `Evaluate` function would probably also be the place to instantiate these variables before executing the action, because we want the terms containing the existential variables to represent facts which are true at the time the conjunct is executed.

# 6. Conclusion

In the previous pages, we have described the first part of the module which is responsible for action execution in our text adventure game. In the introduction, we established the need for additional information about the actions in order for the module's main functor to be able to execute them.

Therefore, in section 2 we showed how we created a database containing this necessary information, by adapting the STRIPS language: we explained why we have to differentiate between two kinds of effects, namely the effects proper and the effects on the player's knowledge of the world around him (section 2.1.2), how we implement in Oz the rules describing the actions (section 2.2), and how we make the rules more powerful by allowing the terms of the precondition and effects lists to be written also in description logic syntax (section 2.3.1), as well as by introducing two functional terms whose value is world-state dependent (section 2.3.2).

In particular, concerning the implementation of the rules in Oz, we showed that the present approach has the advantage that the rules in `ActionDatabase.oz` are lean, meaning that they do not contain any repetitious code, but that instead any repetitions which were necessary for the declaration of the Oz variables in previous versions are now avoided by replacing these variables with integers to indicate co-reference.

Furthermore, we then saw how avoiding having to declare any variables enabled us to simplify the entries and move reoccurring bits of code to more proper places, namely to `InitActions.ozf` for the encapsulation of the rules and to `LookupAction.ozf` for their unification with the input and instantiation.

The consequence of this is that writing new rules is now possible without knowing anything about the way they will later be used by the various functions in `InitActions.ozf`, `LookupAction.ozf` and `Execute.ozf`. Indeed, the only knowledge needed in order to define new actions is now a basic understanding of STRIPS rules' syntax, of description logic and of the meaning of the functional terms we introduced.

After having fairly extensively exposed the whys and hows of the `ActionDatabase.oz`, we then came around to describing how the rules are then made accessible to the lookup function at the start of the game, by being loaded into an Oz dictionary (section 3). At this point, we also went over the function `SubstituteInts`, whose role it is to replace the integers in the rules with variables, and which is written into each encapsulated rule when it is loaded into the Oz dictionary, but applied much later, at lookup time.

Following this, we showed how the actions are looked up in the database during the game: we first described what the input to our module is and how the module's main functor must preprocess this input before it can look up the actions (section 4.1), after which we examined the function `LookupInDB` is some detail (section 4.2), explaining why it uses Oz's search mechanism and what happens within the search predicate, thereby paying special attention to the process of instantiating the rules.

Next, in section 4.3 we went on to explain how we deal with the two possible syntactic variants of the rule terms: having established in section 2.3.1 that, since description logics have a more powerful syntax than STRIPS, the terms used internally by `Execute.ozf` must be in description logic syntax, we then took a look at the way in which the STRIPS terms are converted to description logic syntax.

Lastly, in section 5, we gave an example which showed that evaluating the functional terms from section 2.3.2 just after lookup (within the `Convert` function) is too early and will lead to errors, and we briefly explained how in recent versions of the module the functional terms are now evaluated much later, namely within the call on the function `Executable` in the the main functor itself, although the function `Evaluate` remains for historical reasons in the `LookupAction.ozf` functor. Here we also mentioned which functions in the main functor call up `Evaluate`, but unfortunately, their exact role could not be made clear without a closer look at the main functor, for which we refer the reader to Part II of this documentation, by Ruth Kusterer.

## Appendix: Relaxing the Matching of the Input with the Database Entries

In the footnote on page 10, we mentioned that when we first started out on the project, we originally wanted an input such as `kill(patient:worm1 instrument:sword1)` to unify not only with the entry for `kill(patient:X instrument:Y)`, but also with `kill(patient:X)`. Our intuition was that in the second case, the instrument should simply be ignored, so that it would be possible for the player to add any unnecessary information he wished without preventing a rule from applying, provided all the thematic roles required by the rule were filled.

We also added that in order to achieve this, we had tried using open records in the database entries, i. e. records which do not have a fixed arity, and which are denoted in Oz by "..." in the record, such as in `kill(patient:X ...)`. However, this brought up difficulties at lookup time which we could not pin down, for some of our procedures kept on hanging. Since this was obviously related to the use of open records, because the procedures worked as they should with "normal" (closed) records, we finally gave up the open records, thereby requiring the input to match exactly with the `action` record of the rule in the database.

On 22nd November, 2001, in a personal email to me, Dr. Denys Duchier explained the problem, by revealing that open records in Oz are actually very different from "normal" records:

```
> One thing that is often not clear to people is that open records are
> NOT records: they are constrained variables (denoting records in the
> same sense that a FD variable denotes an integer).  Thus, most record
> operations will block on them, much like arithmetic operations would
> block on FD variables. To constrain open records, you need to use the
> constraints provided by the RecordC module.
```

Following this, I explored two possibilities[15] of re-obtaining the flexibility in unification which we had initially wanted to have. In both cases, some of the functions of our module needed to be reimplemented, using the constraints from the RecordC module (Duchier et al., 2001b).

The first solution involved changing all of the records in the `action` field of the database entries back to open records. In this case, it was the function `SubstituteInts` which hung, and which had to be reimplemented so as not to block on open records, using the following sub-procedure for `Substitute` and a reimplementation of the mapping function which does not block on open records:

[15] I have checked two example files, `lookUpWithOpenRecs.oz` and `lookUpWithOpenRecs2.oz`, into the project's cvs, which illustrate how the two reimplementations presented here respectively work.

```
fun{Substitute Term}
   if {RecordC.is Term} then
      if{Value.isDet Term} then {Record.map Term Substitute}
      else {OpenRecMap Term Substitute} end
   elseif {IsInt Term} then {ToVar Term}
   elseif {Atom.is Term} then Term
   end
end

fun{OpenRecMap OpenRec P1}
   Arity = {RecordC.reflectArity OpenRec}
   NewRec = {RecordC.tell {Record.label OpenRec}}
   fun{Recursion Features}
      case Features
      of nil then NewRec
      [] F|Fs then NewRec^F={P1 OpenRec.F} {Recursion Fs}
      end
   end
in
   {Recursion Arity}
end
```

However, this has the disadvantage that the code responsible for the broader unification is dispersed over three files, namely `ActionDatabase.oz` (the open records), `InitAction.ozf` (the procedure `SubstituteInts`) and `LookupAction.ozf` (the search predicate doing the unification proper).

For this reason, I also tried a second solution, in which the rules remain implemented as closed records in `ActionDatabase.oz`, but are later "opened up" at lookup time by the function `RecToOpenRec`, which is used by a modified search predicate within the function `LookupInDB`:

```
fun {RecToOpenRec Rec}
   Arity = {Record.arity Rec}
   OpenRec = {RecordC.tell {Record.label Rec}}
   fun {Recursion Features}
          case Features
          of nil then OpenRec
          [] F|Fs then OpenRec^F=Rec.F {Recursion Fs}
          end
   end
in
   {Recursion Arity}
end

{Search.base.one
 fun{$}
        Entry = {DBEntry}
        Action = {RecToOpenRec Entry.action}
        NewEntry = {Record.adjoinAt Entry action Action}
 in
        Input = NewEntry.action
        NewEntry
 end}
```

Should the flexible unification we first aimed at ever find its way into a future version of the module, I would personally prefer the second solution to the "open records problem", given that it allows a clear distinction of each functor's role.

## References

Duchier, D., Gardent, C., and Niehren, J. (2001a). Concurrent constraint programming in Oz for natural language processing. University of the Saarland, Department 4.7 General Linguistics, Computational Linguistics. Course notes. Available from `http://www.ps.uni-sb.de/~niehren/Web/Vorlesungen/Oz-NL-SS01/`.

Duchier, D., Kornstaedt, L., Müller, T., Schulte, C., and Roy, P. V. (2001b). System modules, Part II: Constraint programming, chapters 4 and 8 on search engines and feature constraints. Available from `http://www.mozart-oz.org/documentation/system/index`, consulted on 21st October, 2001.

Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208. Cited by Russell and Norvig (1995).

Gabsdil, M., Koller, A., and Striegnitz, K. (2001). Building a text adventure on description logic. In *Proceedings of KI-2001 Workshop on Applications of Description Logics*, Vienna. Available from `http://www.coli.uni-sb.de/~kris/papers/dlws.ps.gz`.

Haarslev, V. and Möller, R. (2001). Racer user's guide and reference manual. Version 1.6, Computer Science Department, University of Hamburg, Germany. Available from `http://kogs-www.informatik.uni-hamburg.de/~race/racer-1-6-manual.pdf`.

Ingargiola, G. (1995). Class notes for *CIS587: Introduction to Artificial Intelligence*. Available from `http://joda.cis.temple.edu/~ingargio/cis587/readings/wumpus.shtml`, consulted on 2nd October, 2001.

Lormand, E. (1999). The frame problem. In *The MIT Encyclopedia of the Cognitive Sciences*. Available from `http://www-personal.umich.edu/~lormand/phil/cogsci/frame.htm`.

Nilsson, N. J. (1998). *Artificial Intelligence: A New Synthesis*, chapter 22, pages 373–375. Morgan Kaufmann, San Francisco, California.

Russell, S. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*, chapter 11.4: Basic Representations for Planning, pages 343–345. Series in Artificial Intelligence. Prentice Hall, Englewood Cliffs, NJ.

Wahlster, W. (2001). Einführung in die Methoden der KI (Introduction to the methods of artificial intelligence). Departement 6.2 Computer Science, University of the Saarland. Course notes. Available from `http://w5.cs.uni-sb.de/ss02/materialien/`.